Automatic Parallelism Exploitation for FPL-based Accelerators

Jürgen Becker  
Darmstadt University of Technology  
Institute for Microelectronic Systems  
Karlstr. 15,  
D-64283 Darmstadt, Germany  
becker@mes.th-darmstadt.de

Karin Schmidt  
Daimler-Benz AG  
Research and Technique  
Wilhelm-Runge-Str. 11,  
D-89013 Ulm, Germany  
karin.schmidt@dbag.ulm.daimlerbenz.com

Abstract  
The paper introduces the first complete programming framework for coarse grain dynamically reconfigurable accelerators and their application development. It includes a general model for cooperating host/accelerator platforms and a parallelizing compilation technique derived from it. The paper is an introduction illustrating these techniques and their principles by examples: a machine architecture (briefly), and its application development framework performing a “software-only” accelerator implementation (synthesis). The paper’s focus is to explain the exploitation of four different levels of parallelism during this compilation process for achieving optimized speedups and hardware resource utilization.

1. Introduction  
Although the demand for high throughput is rapidly growing, the parallel processing scene is in a crisis and many vendors went out of business. Here the still mainly unsolved problem is the very late binding of communication channels (figure 1), which causes massive fine granularity switching overhead at run time. Immense design manpower is recruited for hardwired parallelism on an exploding number of types of add-on accelerator circuits and boards with shrinking product life times. Thus, a paradigm shift from designing customized (sub) systems on chips (or boards) to more standardized platforms: to “programming” dynamically reconfigurable circuits, representing a shift from hardware to (structural) “software”. Reconfigurable platforms lead to implementations avoiding the massive run time switching overhead known from concurrent platforms in migrating it to compile time (figure 1). But with reconfigurable platforms available today, and mainly used for glue logic, such a paradigm shift is not likely to happen. Their single-bit fine granularity parallelism is insufficient in time and area. We need reconfigurable platforms supporting coarse granularity parallelism [5]. This paper introduces a novel methodology including such a platform implemented by the first parallelizing programming framework for coarse grain dynamically reconfigurable accelerators.

Such novel design methodologies seem also to be urgently needed for architecture and parallelism exposition on different levels, because the development of new chip-technology generations is no endless spiral, since there exist physical as well as financial borders.

Hardware has become soft. Emanating from the technology of field-programmable logic (FPL) and its application the awareness of the new paradigm of structural programming is growing. A class of commercially available FPL circuits makes use of RAM-based reconfigurability, where functions of circuit blocks and the structure of their interconnect is determined by bit patterns having been downloaded into “hidden RAM” included in the circuit. Modern FPL circuits are reconfigurable within seconds or milliseconds, even partially or incrementally. Such “dynamically reconfigurable” circuits may even reconfigure themselves: An active section of the circuit programs an idling other section of the same circuit. So we now have two programming paradigms: programming in time and programming in space. We distinguish two kinds of “software”:

- sequential software (to derive code downloaded to RAM)
- structural software (code downloaded to hidden RAM)

Tools like XACT and others are a kind of compilers or assemblers accepting structural software. The R&D area of Custom Computing Machines (CCMs: [10]), such as FPGA-based CCMs (FCCMs [3] [4]), merge the use of both, sequential and structural software, into a technique to speedup performance. Very often such CCMs are configured as accelerators supporting a host, like e. g. a PC or workstation.

Such applications including a CCM are dual software-only implementations: traditional (procedural) software running on the host, together with structural software running on a reconfigurable accelerator co-processor. A number of experimental systems having been proposed or implemented derive both kinds of code, sequential code and configuration code, from the same source, such as e. g. in C language. This means, that both, sequential and structural software, may be written in the same language.

But currently - in contrast to software engineering - ”structural programming“ usually is much more difficult and requires highly experienced hardware experts for a number of reasons. This paper introduces an approach, which makes software-only accelerator implementation much easier with the goal to provide a universal programming interface for general purpose application development.

The problems to be solved by this approach stem from hardware/software codesign, performance analysis and parallelizing compilers to exploit inherent program parallelisms at different levels to reduce total design time as well as to utilize the
hardwired ASIC reconfigurable v. Neumann
before fabrication within fabrication at compile time at run time
early communication path binding time late

Figure 1. Influence of target technology platforms

The hardware gap. Contemporary FPGA-based accelerator architectures are far from being general purpose platforms. FPGAs currently available commercially have several severe drawbacks:

• FPGAs are by far too area-inefficient,
• FPGAs have been designed for random logic only,
• FPGAs are too fine grain and too slow, and
• FPGAs provide a poor cost/performance ratio for highly computing-intensive applications.

Using reconfigurable platforms currently available commercially is mainly based on tinkertoy approaches at glue logic level. No general common model has been developed so far. We need a new class of reconfigurable hardware capable to support a novel basic computational paradigm of structural programming for software-only accelerator implementation of high performance applications.

The software gap. The area also suffers from a software gap: only a few application development tools are available, which are difficult to use. To much hardware expertise, and even also routing and placement expertise is needed for structural software implementations. We need much more powerful application development support environments, like compilers accepting application problems expressed in programming languages, like C or Java. But such structural software compilers are currently not available.

The modeling gap. Another problem is the wide variety of architectures urged by optimization needs which stem from these drawbacks. Neither in CCMs nor in hardware/software codesign a common model is available. The von Neumann paradigm does not efficiently support “soft” hardware because of its tight coupling between instruction sequencer and ALU [11]. You need a new instruction sequencer, as soon as the data path is changed by structural programming: the architecture falls apart.

The education gap. Most Computer Science curricula do not create awareness, that hardware has become soft, nor, that hardwired, structural and sequential software are alternatives to solve same problems. Using the technology of field-programmable logic (FPL) as basis of the paradigm of structural programming should be included in academic main courses.

The paper is organized as follows: first the target hardware platform is briefly sketched for comprehensibility. In section 3 the parallelism exploitation on four different levels (task-, loop-, statement-, and operation-level) is explained, when the proposed partitioning compilation techniques for FPL-based accelerators are applied. Finally, obtained results are discussed with a smoothing example in section IV.

2. The Target Hardware Platform

A new kind of structurally programmable technology platform is needed. To support highly computing-intensive applications we need structurally programmable platforms providing word level parallelism instead of the bit level parallelism of FPGAs. We need FPAAs (field-programmable ALU arrays) instead of FPGAs. For flexibility to implement applications with highly irregular data dependencies, each PE (processing element) should be programmable individually. Each PE should be a reconfigurable ALU (rALU). Also the local interconnect between particular PEs or rALUs should be programmable individually. A good solution is the Kress Array [12], [15]. It permits to map highly irregular applications onto a regularly structured hardware platform. To realize the integration of such soft ALUs into a computing machine, a deterministic data sequencing mechanism is also needed, because the von Neumann paradigm does not efficiently support “soft” hardware [9]. As soon as a data path is changed by structural programming, a von Neumann architecture would require a new tightly coupled instruction sequencer. A good backbone paradigm for implementing such a deterministic reconfigurable hardware architecture based on data sequencing is the procedurally data-driven Xputer paradigm, already published many times in the past [13]. For further details about this paradigm and built Xputer prototypes please see the given literature. The Xputer paradigm conveniently supports soft ALUs like in the rALU array concept (reconfigurable ALU array, see figure 2), and it bridges the modeling as well as the education gap.

Based on this accelerator structure a general purpose FPL-based accelerator architecture can be realized by integrating several Xputer-based accelerators to modern microprocessors, and to partition the given workload of application programs to be mapped onto this target hardware platform in an optimized way among the microprocessor and its accelerator(s) [2], [8], [11].

3. Parallelism Exploitation by Partitioning Compilation Techniques

Structural software being really worth such a term would require a source notation like the C language and a compiler which automatically generates structural code from it. For such a new class of accelerator hardware platforms a completely new class of compilers is needed, which generate both, sequential and structural code: partitioning compilers, which separate a source into two types of cooperating code segments:

Figure 2. Illustrating the Kress Array Principles.
3.1 Profiling-driven Host/Accelerator Partitioning for Exploiting Task Level Parallelism

The profiling-driven first level of the dual CoDe-X partitioning process is responsible for the decision which task should be evaluated on Xputer-based accelerators and which one on the host. Generally, four kind of tasks can be determined:

- **host tasks** (containing dynamic structures,
- **Xputer tasks** (candidates for Xputer migration),
- **MoPL-code segments** included in X-C source, and
- **generic Xputer library function calls**.

The **host tasks** have to be evaluated on the host, since they cannot be performed on Xputer-based accelerators. This is due to the lack of an operating system for Xputers for handling dynamic structures like pointers, recursive functions etc., which can be done more efficiently by the host. The generic Xputer **library functions** and included **MoPL-code segments** are executed in any case on the accelerator. All other tasks are **Xputer tasks**, which are the candidates for the iterative first level partitioning step determining their final allocation based on simulated annealing [2], [8], [11]. The granularity of **Xputer tasks** depends on the hardware parameters of the current accelerator prototype, e.g. the maximal nesting depth of for-loops to be handled by data sequencers, the number of available PEs within one rALU array, etc.

For all generated tasks corresponding host and/or Xputer performance values are determined, which are used in each iteration of the first level partitioning process for evaluating the complete application execution time, which represents the optimizing goal of this process. For details about performance evaluation in CoDe-X see [2], [7]. Since this host/accelerator partitioning methodology should be independent from an Xputer hardware prototype version, a hardware parameter file controls the
partitioning. Thus, CoDe-X partitioning strategies and algorithms can be used also for future prototypes.

Based thereon, the first level partitioning method of CoDe-X can be subdivided into the following major steps:

- a compiler front end performs a syntactical and semantical analysis of the X-C input program including also the verification of correct included MoPL-3 code segments as well as of generic Xputer-library function calls by using a MoPL-3 parser,
- a set of tasks is derived from the resulting program graph \( G \) (four kinds of tasks), whereas the tasks’ granularity depends on given Xputer hardware parameters,
- a data flow analysis step identifies inter-task data dependencies, resulting in inter-task communications, e.g. memory re-mappings. Output is a task graph representing all task’s control/data dependence relations,
- based on the hyperplane concurrency theorem by Leslie Lamport [16], optimizing code transformations are performed to appropriate Xputer tasks, for exploiting potential intra-task code parallelism and to optimize the

```
for (i=0; i<477; i++)
for (j=0; j<357; j++) {
    w[i+1][j+1] = (w[i][j] + w[i+1][j] + w[i+2][j] + w[i][j+1] + w[i+1][j+1] + w[i+2][j+1] + w[i][j+2] + w[i+1][j+2] + w[i+2][j+2]) / k;
}
```

```
parbegin
for (i=0; i<238; i++)
for (j=0; j<357; j++) {
    w[i+1][j+1] = (w[i][j] + w[i+1][j] + w[i+2][j] + w[i][j+1] + w[i+1][j+1] + w[i+2][j+1] + w[i][j+2] + w[i+1][j+2] + w[i+2][j+2]) / k;
}
```

```
for (i=239; i<477; i++)
for (j=0; j<357; j++) {
    w[i+1][j+1] = (w[i][j] + w[i+1][j] + w[i+2][j] + w[i][j+1] + w[i+1][j+1] + w[i+2][j+1] + w[i][j+2] + w[i+1][j+2] + w[i+2][j+2]) / k;
}
```

```
parend
```

Figure 5. Strip mining: smoothing algorithm
• **loop interchanging** transformation of a pair of loops by switching inner and outer loop, without affecting the outcome of the loop. This supports possible vectorization of the inner loop, and/or parallelization of the outer loop.

The fifth transformation technique implemented by CoDe-X' 1st level partitioner doesn’t require a data flow analysis according to the adapted hyperplane theorem and is called:

• **loop unrolling**: increasing the step width of single loop iterations and decreasing therefore the total number of iterations, by unrolling the loop body up to a certain factor. This technique is applied for increasing performance and optimizing the accelerator’s hardware utilization, e.g. reducing the number of idle DPUs within a rALU array.

The transformation technique to be explained more detailed in this paper is **strip mining** (see figure 5), which exploits parallelism on task level, given within X-C input programs. For examples, applications and parallelism exploitation of the other above introduced code transformation techniques please see [2]. **Strip mining** makes it possible to compute these chunks concurrently on different Xputer-based accelerator modules. In figure 5 the **strip mining** application of a computation-intensive loop nest within an image smoothing algorithm (see also application example in section 4) is shown. In our approach the block size of the chunks depends on hardware parameters describing the current accelerator prototype (e.g. the number of available accelerator modules) in order to achieve an optimized performance/area trade-off. This technique can be used e.g. often in image processing applications by dividing an image in stripes of equal sizes in order to manipulate these stripes concurrently on different accelerator modules [8], whereas each stripe execution represents one Xputer task.

**Scheduling, Synchronization, and dynamic Reconfiguration** (s), The Xputer Run-time System (XRTS) [18] provides the software-interface from the host to Xpreter-based accelerators. The main purpose of this system is controlling and synchronizing applications executed on the accelerator, but additional features like program debugging are also possible. The Xputer Run-time System can be used interactively or inside a host process and has following features:

• XRTS loads Xpreter object files (sequential/structural code for data sequencer(s)/rALU Array)
• XRTS loads application data binary files with optimized distribution to the memories of different modules to minimize inter-module communication

The **run time scheduling process (RTS-process)** for performing task scheduling, communication/synchronization and reconfiguration(s) during run time activates the XRTS and is generated by CoDe-X' 1st level partitioner after the final task allocation and scheduling is determined. The meta-code template of such an **RTS-process** is given in figure 7.

Before explaining this **RTS-process**, the **task scheduling** step is described first. This step determines a deadlock-free total execution order of all allocated tasks. Parallel task execution and reconfiguration are possible, according to the detected inter-task data dependencies. Therefore, a static resource-constrained list-based scheduling technique is applied by building a **task priority list** for each hardware resource, e.g. the host and its connected accelerator modules. A task is inserted into the **priority list** of its allocated hardware resource according to a two-stage priority function, which is explained below. Since this scheduling problem is NP-complete, appropriate heuristics have been selected for finding a good solution.

The **list-based scheduling algorithms** [19] belong to the heuristic methods. Since it is a very flexible scheduling method, it is described in detail in literature. In our case, a priority function for building a **priority list** is used to choose from all candidate tasks the tasks to schedule next. The list-based scheduling algorithm in our approach tries to minimize total execution time under resource constraints, and is applied as follows (see also [2]):

The heuristic **priority function** for building the **priority lists** is introduced, since it is an important issue of the proposed scheduling method:

• **inter-task data dependencies** have the highest priority; for each hardware resource the list of its allocated tasks is pre-ordered according to these determined data dependencies, which results in a partial task execution order, and guarantees consistency of data dependent task executions, and

• for second priority criteria several possibilities exists [2]:
  • the easiest and most straightforward way is to order the priority queues according to each task’s determined ASAP- or ALAP-synchronization point in relation to the program’s data flow, e.g. **shortest ASAP first** or **shortest ALAP first**, or
  • an related alternative is to take each task’s **mobility metric** as criterion, computed as the difference of determined ASAP- and ALAP-synchronization points, e.g. **shortest mobility first**. Thus, tasks with tight data dependency constraints are scheduled first, which may handicap the execution of other tasks, or
  • taking each task’s **critical path location** within the **task graph** as priority criterion, which is probably the most sophisticated choice in our case and implemented in the current CoDe-X version. The idea here is to schedule tasks first that have many and/or computation-intensive data dependent tasks to be executed thereafter, because this may be critical for the total application execution time

Examples of different **critical path locations** within a **task graph** $TG$ are illustrated in figure 6. The numbers at the upper right corner of all tasks represent some pre-computed fictitious task execution time values for host or accelerator execution, dependent on their allocation.

Based on above introduced assumptions, the static resource-constrained list-based scheduling technique is implemented in the CoDe-X framework in building a task **priority list** for each hardware resource (host and Xptron-based accelerators) according
Additionally, for each task of task allocation and scheduling information, as described above, generated task sequence of all tasks, which includes the complete task graph, is terminated with an optimized result for the partitioning loop, in each iteration of CoDe-X' profiling-driven reconfiguration. This ASAP reconfiguration technique results in a possible 

First, a total task list (TASK_LIST) is initialized with the generated task sequence of all tasks, which includes the complete task graph, and scheduling information, as described above. Additionally, for each task of TASK_LIST a counter (DEPEND_CT(Task_i)) is initialized with the number of dependent tasks, on which this task’s data depends (DEPEND(Task_i)). The constant TASK_NUM contains the total number of all tasks.

The next step is to start execution of all executable tasks, which are scheduled first in their static priority lists (Schedule_List_i), see above introduced resource-constrained list-based scheduling technique. Therefore, the first task of each priority list is taken (FIRST(Schedule_List_i)) and stored into First_Task. If their corresponding counters DEPEND_CT(First_Task) are zero, they are safely executable, because no data dependent task(s) has (have) to be finished before. The function CONFIGURE(First_Task) initiates the configuration of First_Task, and START(First_Task) triggers the allocated hardware resource to start the task. The constant HW_RES_CT contains the total number of available hardware resources, e.g. the host and connected accelerator modules.

The next nested for-loop meta-code segment in figure 7 represents the main part of the RTS-process performing a busy waiting loop looking for task terminations. After the XRTS signals one task termination, the RTS-process updates all relevant counters holding the actual number of data dependent tasks to be finished, before starting the corresponding task. Therefore, the function WAIT_TASK_END() performs a busy waiting loop, until XRTS signals a task termination, and saves the terminated task into Terminated_Task. Next, the function DEPENDENT_ON(Terminated_Task) computes the set of all tasks, which are directly data dependent on Terminated_Task, corresponding to an arc in the task graph. The counter DEPEND_CT(Task) of each task within this set is decremented, and if DEPEND_CT(Task) equals zero, the corresponding task is executable and marked as ‘ready’ (READY(Task)).

The following for-loop within the main loop nest of the RTS-process takes from the priority list of the terminated task’s hardware resource (HW_RES_NUM(Terminated_Task)) the task to be scheduled next (First_Task), which is the first task in the corresponding static priority list (FIRST(Schedule_List_i)), and start its configuration (CONFIGURE(First_Task)). This can be done in advance of being executable, because the taken task is executed in any case next on the allocated hardware resource. This ASAP reconfiguration technique results in a possible reduction of the complete application execution time. If the viewed task is additionally executable (marked ‘ready’) its data files are updated by UPDATE_DATA(First_Task), realizing the inter-task communication by updating all shared variables’ values of First_Task with data values of finished tasks, on which First_Task depends (for more details see [2]).

### 3.2 Resource-driven Partitioning for Exploiting Statement Level Parallelism

The X-C compiler [20] realizes the 2nd level of partitioning and translates accelerator migrated X-C subset code segments into code which can be executed on the Xputer. It divides the accelerator source code into sequential code for the data sequencer and structural code for the RALU array. The compiler performs a data and control flow analysis. The Xputer hardware structure itself provides best parallelism at statement or expression level. Exploiting statement level parallelism [20] deals with the fundamental problems similar to those in compiling a program for parallel execution on a multiprocessor system. These problems are: (1) Identify and extract potential parallelism, (2) partition the program into a sequence of maximal parallelized execution units according to the granularity of the architecture and the hardware constraints, (3) compute an efficient allocation scheme for the data

![Task Graph TG](image)

**Figure 6. Different critical path locations CPL(T_i) of task graph TG**

to the following criteria:

- first criteria: data dependencies (consistency), and
- second criteria: critical path location (see figure 6).

Additionally, each task’s data dependency information, given by the task graph, is used in the below explained RTS-process for guaranteeing a consistent concurrent task execution within the total execution order. The introduced resource-constrained list-based scheduling technique generates as output the completed task sequence incl. the task scheduling information, and is applied in each iteration of CoDe-X' profiling-driven first level partitioning loop after the task allocation step. The static task scheduling information is necessary for total application execution time estimation in each iteration, being the cost function to be minimized during host/accelerator partitioning.

List-based scheduling algorithms are widely used in synthesis systems since they are simple to adjust on given problems with appropriate priority functions. For small examples, solutions have been shown that the algorithm differs not much in execution times from the optimal ones [6].

In the following the necessary RTS-process is described, which coordinates the realization of the determined final task allocation and scheduling by handling the necessary dynamic synchronization, communication and reconfiguration. This RTS-process coordinates the concurrent execution of all CoDe-X generated tasks, which are the result of partitioning an application into procedural and structural software. The RTS-process runs on the host computer and is generated after the final task allocation and task scheduling is performed, e.g. after CoDe-X' first level partitioning loop is terminated with an optimized result for the complete application execution time. The meta-code of the RTS-process solution implemented in the current CoDe-X version is given in figure 7.

First, a total task list (TASK_LIST) is initialized with the generated task sequence of all tasks, which includes the complete task allocation and scheduling information, as described above. Additionally, for each task of TASK_LIST a counter (DEPEND_CT(Task)) is initialized with the number of dependent tasks, on which this task’s data depends (DEPEND(Task)). The constant TASK_NUM contains the total number of all tasks.
/* Initialization of total task list, and for each task a counter holding number of data dependent tasks */
TASK_LIST = T;
for (i = 1; i <= TASK_NUM; i++) {
   DEPEND_CT(Task_i) = DEPEND(Task_i);
}
/* starting all first executable tasks on each hardware resource HW_Res */
for (i = 0; i <= HW_RES_CT; i++) {
   First_Task_i = FIRST(Schedule_List_i);
   if DEPEND_CT(First_Task_i) = 0 {
      CONFIGURE(First_Task_i);
      START(First_Task_i);
   }
}
/* scheduling loop waits for task terminations, updating relevant counters and mark executable tasks 'ready' */
for (i = 0; i <= TASK_NUM; i++) {
   Terminated_Task = WAIT_TASK_END();
   foreach Task in
      DEPENDENT_ON(Terminated_Task) {
         DEPEND_CT(Task) = DEPEND_CT(Task) - 1;
         if DEPEND_CT(Task) = 0 {
            READY(Task);
         }
      }
}
/* taking from priority list of terminated task's hardware resource the task to be scheduled next, and start its configuration. If it is executable (marked 'ready'), its data files are updated, and its execution is started */
i = HW_RES_NUM(Terminated_Task);
First_Task_k = FIRST(Schedule_List_k);
CONFIGURE(First_Task_k);
if First_Task_k = 'ready' {
   UPDATE_DATA(First_Task_k);
   START(First_Task_k);
}
/* Figure 7. CoDe-X’ Run time scheduling process (RTS-process) controlling task synchronization, communication and reconfiguration */
in the Xputer data map, and (4) generate efficient and fast code.

First a theory is needed for the program partitioning and restructuring. Result of this step is the determination of a partial execution sequence with maximal parallelized execution units. Secondly the program’s data has to be mapped in a regular way onto the 2-dimensionally organized Xputer Data Memory, followed by a computation of the right address accesses (data sequencing) for each variable. Code generation for Xputer-based accelerators results in structural code for the configuration of the rALU array, and more sequential code containing the parameter sets for the multiple data sequencers.

**Determination of a Program Execution Sequence.**

Starting point here is the graphical program representation G = (N, E) with node set N and arc set E. The control flow of the program is partitioned first in order to find parallelizable subgraphs, which is followed by a data partitioning by partial vectorization based on the level-k-dependence graph [23].

**Partitioning of the Control-Flow.** Given is the program graph G = (N, E). The node set N has to be partitioned, resulting in a number of subgraphs G_k, 1 ≤ k ≤ n, and an arc set E*, defining a partial execution order. The partitioning function is given as:

\[ \pi : N \rightarrow N_i, \text{ with } N_i \cap N_j = \emptyset \text{, for } i \neq j. \]

This function computes a number of subgraphs:

\[ G_k = (N_k, E_k^k), \text{ with node set:} \]

\[ N_k = \{ (n \in N)|((\pi(n) \rightarrow N_k) \} \text{ and the arc set:} \]

\[ E_k^n = \{((n_i, n_j) \in E)|(\pi(n_i) \rightarrow N_k \land \pi(n_j) \rightarrow N_k) \} \]

The arcs of the cut set E* given by the original set E and all sets:

\[ E_k^* \text{, defined by } E^* = E \cap \bigcup_{k=1}^{n} E_k^k \]

then give the partial execution order “≤”. For the structure of the subgraphs an additional criterion has to be formulated, namely that each subgraph has to be convex. The question is now, how the partitioning of a graph into convex subgraphs can be achieved. This is performed by specifying an equivalence relation on the node set N resulting in corresponding equivalence classes which represent convex subgraphs. The definition of a partial order relation for the equivalence classes then gives the execution sequence. The **Equivalence Relation Connect** specifies an equivalence relation in the set of nodes N by:

\[ a \sim b \Longleftrightarrow \pi(a) = \pi(b) \text{, with } a, b \in N . \]

Connect is reflexive, symmetric, and transitive and partitions the node set N into disjoint, non-empty equivalence classes:

\[ [a]_\sim = \{ b | (b \in N \land a \sim b) \} \]

The set of equivalence classes, the quotient of N by connect is:

\[ N / \sim = \{ [a]_\sim | (a \in N) \} . \]

In each equivalence class \([a]_\sim\) are the nodes of N which are assigned later to one Xputer execution unit. The node sets given by \(N_k\), 1 ≤ k ≤ n, correspond to the associated equivalence classes of connect and thus represent the contents of the execution units. A partial order relation defines the partial execution order of the equivalence classes. A partial order relation sequence over N / \sim is defined by:

\[ [a]_\sim \leq \text{ sequence } [b]_\sim \Longleftrightarrow \text{ path}(a, b) \]
The partial order relation sequence corresponds to the set of arcs $E'$. The presented method for control flow partitioning results in a coarse grained sequence of units with a partial execution order. The units are still target-hardware independent.

**Partitioning of the Data-Flow.** The goal of the next compilation step is to maximally parallelize each of the determined blocks in the sequence. This gives the basis for a good exploitation of the available hardware resources. First the level-1-dependence graph [23] for each of the blocks is build. Since all kinds of index expressions in array variables are allowed (zero, single, and random index variable) a hierarchical framework of according tests is needed to determine flow, anti, or output data dependences [23] together with the level where the dependences exist. The level-1-dependence graph is subdivided into convex subgraphs using the method introduced in the previous section. Each of the subgraphs is then maximally vectorized by using the Allen-Kennedy Vectorization Algorithm. Result is a new sequence of maximally parallelized units containing a partial execution order. Vectorization then generates a maximum degree of parallelism in a statement block of a loop nest for statements having no dependences or being not part of a recurrence or member of a cycle [14]. But this kind of parallelization is performed independent of any resource constraints. This would surely violate the Xputer hardware constraints, since the execution cannot be realized in a pipelined mode like in a vector computer. Therefore a hardware-dependent vectorization factor is introduced. This factor is responsible to subdivide the generated vectors into smaller parts such that an optimal target hardware exploitation can be achieved.

**Vectorization Factor $VF_j$:** Given is a normalized loop $J$ with lower limit $l_j$ and upper limit $u_j$ and a statement $S$ with a variable $a[f(J)]$, $f(J)$ is an index function. The loop has been vectorized by replacing the index function $f(J)$ by $f(J) : f(u_j)$. This vector has to be partitioned by the introduction of a hardware-dependent vectorization factor $VF_j$. The factor has to be chosen such that (1) $VF_j \in \mathbb{N}$ and (2) $(f(u_j) - f(l_j) + 1) \mod VF_j = 0$.

The vectorization factor $VF_j$ has to divide the upper limit of a loop index function $f(u_j)$ minus its lower limit $f(l_j)$ by itself. Introducing a vectorization factor means that (1) the step widths of the according loops have to be adapted (“Do $J = l_j$ to $u_j$ by $VF_j$”) and (2) the index function in the variable has to be changed by the number of accessed variables at one time step (“a $f(J)$: $VF_j$”), e.g. $A_i = i+1, j=1:10 = C_i, j=2:10 / 2 + C_i, j=1:10$. Each vectorized loop $L_j$, $1 \leq j \leq m$, may have its own vectorization factor $VF_j$. For all variables which are contained in the same vectorized loop the same vectorization factor has to be used. Partial vectorization together with the concept of the vectorization factor transforms each of the former coarse-grained blocks $B_{jk}$, $1 \leq k \leq m$, into a new sequence of parallelized blocks. Each of the blocks provides the special Xputer granularity and tries to optimally exploit the target hardware resources. This is the key for achieving a high performance during program execution.

For our smoothing algorithm the strong SIV test [20] has been applied for all dependences in the statement with the result that the second loop (for $y$) may be vectorized with a step-width of 3. This results in the use of 27 DPUs within the rALU array (see figure 8).

**Data Mapping and Data Aligning.** The next step in compilation is to decide how the program data (variables, arrays, ...) can be mapped onto the two-dimensionally organized Xputer Data Memory $DM = \{DM_x, DM_y\}$ in a regular fashion, and how the data fields of differently mapped data variables can be aligned to a combined data field in order to use only data sequencer (one scan pattern).

The 2-D Data memory $DM$ is in contrast to the defined arrays which are of higher dimensions. This leads to the mapping problem resulting in the definition of a data allocation scheme. This leads to the data alignment problem. Unrolling the dimensions $I$ of a variable $A$ defined to be $d$-dimensional, $d > 2$, and $d \in \mathbb{N}$, means to determine a function $dnmap$ from the index domain of the associated data object to the two-dimensional index domain of an Xputers data map $DM$, by $dnmap$:

$$I_i \rightarrow \{DM_{x_i}, DM_{y_i}\}, \quad 1 \leq i \leq n.$$  

The question is what realization strategy may be chosen for $dnmap$. First the dimensions in the array definition are numbered from the right to the left and are then mapped. Even numbers are mapped onto the x-coordinate ($DM_x$), odd numbers onto the y-coordinate ($DM_y$) of the Xputers data memory $DM$. Xputer dimension mapping is a kind of planarization. Due to the restricted space, see [20] for details of Data Mapping and Aligning.

**Determination of the Data Sequencing.** The accessing of the program data variables by their indices is needed for the generation of scan patterns from which the parameter sets for the data sequencer and the instruction sequencer are computed. This results in the determination of an access sequence for each variable according to their indices together with their data fields which have been mapped into a two-dimensional form. The access sequences then can be used for a computation of corresponding scan patterns and parameter sets. The computation of access sequences is influenced by mapping, alignment, index expressions, and according loop limits (upper, lower, step width).

```cpp
// ALEX-File
rALUsubnet SubNet_0 (SW_0_0)
ScanWindow SW_0_0
{
  int SW1_2_4 at [2][4];
  int SW1_1_4 at [1][4];
  int SW1_0_0 at [0][0];
  HandleOffset [0][3];
}

// SW1_0_0 = (((((SW1_0_0 + SW1_1_0) + SW1_2_0) + SW1_0_1) + SW1_2_1) + SW1_0_2) + SW1_1_2) + SW1_2_2) / 9

// SW1_0_1 = (((((SW1_0_1 + SW1_1_1) + SW1_2_1) + SW1_0_2) + SW1_1_2) + SW1_0_3) + SW1_1_3) + SW1_2_3) / 9

// SW1_0_2 = (((((SW1_0_2 + SW1_1_2) + SW1_2_2) + SW1_0_3) + SW1_1_3) + SW1_0_4) + SW1_1_4) + SW1_2_4) / 9
}
```

// MoM-Assembler

Figure 8. Generated ALEX description for the smoothing algorithm (vectorized)
For reasons of the two-dimensional Xputer data map the hardware of a data sequencer is implemented such that it generates simultaneously address parts for the y-address and for the x-address. Hence the inner part of an access sequence consists of two values for the lower limit \( (y_{\text{lower}}, x_{\text{lower}}) \), two values for the upper limit \( (y_{\text{upper}}, x_{\text{upper}}) \), and two for the step widths \( (\text{step}_y, \text{step}_x) \). The computed values directly correspond to a fast scan pattern which is called videoscan \[13\]. Since a loop nest for a variable \( a \) can be of n-dimensional form “for \( i_m = l_m \) to \( u_m \)”, this base parameter set has to be changed after each completion of a videoscan. For more details of data sequencer parameters see \[20\].

**Structural and sequential code generation.** For each block out of the generated sequence according scan windows and access sequences have been generated, which can be now transformed into Xputer code for the accelerator hardware. According to the hardware structure of an Xputer it can be distinguished between structural code for the configuration of the rALU array \[15\] and sequential code containing the parameter sets for the Xputer’s data sequencers. The structural code file only contains statements with arithmetical and logical expressions and no more control statements (see figure 8 with the ALEX (arithmetical and logical expressions for Xputer) description for the smoothing filter. The ALEX description file serves as input to the data path synthesis step (DPSS) to exploit operation level parallelism. The DPSS applies loop folding to inner loops, which performs the pipelining of loop iterations (loop level parallelism). If the DPSS receives vectorizable code from the 2nd partitioning level (see section 3.2), the corresponding statements can be executed in parallel (statement level parallelism). Additionally, the DPSS can pipeline vectorizable statements within the rALU array in executing them in a PE-pipeline (operation level parallelism), if not enough PEs are available for executing them in parallel. For further details about the steps of the DPSS see \[12\], \[15\].

The sequential file includes the parameter sets for the multiple Data sequencers (e.g.: scan pattern description, and code for the instruction sequencer, the scan windows’ access strategies, and the rALU array activation. The first part of the file initializes the numbers of the rALU array subnets and the numbers of address generators within the data sequencer. The stack size is initialized, the segment sizes of the data-fields are given, and the status of the IS and according macros are defined. Further details and examples can be found in \[20\].

**4. Examples: Image Processing Applications**

To illustrate the parallelism exploitation of the introduced compilation techniques by this paper the already used image smoothing algorithm is summarized in this section. Smoothing operations are used primarily for diminishing spurious effects, that may be present in a digital image as a result of a poor sampling system or transmission channel. The effect of a smoothing filter is shown in figure 9. Given an \( N \times N \) image \( f(x,y) \), the procedure is to generate a smoothed image \( g(x,y) \), whose gray level at each point \( (x,y) \) is obtained by averaging the gray-level values of the pixels of \( f \) contained in a predefined neighborhood (kernel) of \( (x,y) \). In other words, the smoothed image is obtained by using equation \(1\):

\[
g(x, y) = \frac{1}{M} \sum_{(n, m) \in S} f(n, m) \quad \text{(eq. 1)}
\]

for \(x,y = 0,1,..., N-1\). \( S \) is the set of coordinates of points in the neighborhood of (but not including) the point \((x,y)\), and \(M\) is a pre-computed normalization factor.

The tasks containing I/O routines for reading input parameters and routines for plotting the image are executed by the host, because they cannot be executed on the Xputer. The remaining tasks are potential candidates for mapping onto the accelerator. For more details about partitioning of such image processing applications and their complete task structure please see \[2\]. One computation-intensive task of these potential migration candidates has been taken in section 3.1 - section 3.2 as example task for applying the introduced compilation techniques for FPL-based accelerators. Parallelism at task level was exploited in applying the...
code transformation *strip mining*, resulting in several tasks to be executed concurrently on different accelerator modules (see section 3.1). To the inner loop of each of these tasks vectorization has been applied (see section 3.2), resulting in parallelism at statement level (if enough hardware resources, e.g. PEs within rALU array are available), or parallelism by *pipelining at operation level* (if not enough PEs are available for statement parallelization [12], [15]). Additionally, *loop folding* (see section 3.1) can be applied for achieving parallelism by pipelining at loop level.

The DPSS mapping of one vectorized task (3 vectorized statements of inner loop, see figure 5 and figure 8) incl. the time scheduling of corresponding rALU array operators is shown in figure 10. Hereby, the $\Sigma$-operator stands for summing up 3 subsequent pixel values $w_{ij}$. The derived structure takes the values of the $w_{i3}$ pixel column as input to the sum operators and emits the new value for $w_{i1}$ from the division operator. The values of the columns $w_{i3}$ and $w_{i1}$ need not to be read, as they are stored inside the sum operators from the previous steps. The next step of the DPSS is the data scheduling of necessary operands. The sequencing graph in figure 10 contains also this scheduling information, whereas 2 operands $w_{i1}$ can be brought over 2 parallel busses to PEs of the rALU array. For demonstration purposes, we assume a simplified timing behavior for our filter example, where an I/O operation has a delay of one time step, an addition or sum operator two time steps and a division by the smoothing constant three time steps. For more details and examples of DPSS mappings see [15].

5. Conclusions

A novel accelerator methodology for coarse grain dynamically reconfigurable computing machines has been outlined being drastically more area-efficient than using FPGAs and other fine granularity platforms. The general model for the underlying hardware has been introduced briefly, incl. its flexible word-oriented ALU structure. The coarse grain reconfigurable ALU array provides instruction level parallelism exploited by the optimized operator mapping and scheduling tool DPSS.

Based thereon, the accelerator compilation framework CoDe-X implements a two-level partitioning method for such a target hardware structure without requiring hardware experts, and providing a universal programming interface. This software-only accelerator implementation method accepts C language programs (as well as optionally included hand-honed data-procedural application parts), and generates *sequential software* for the host, as well as *

<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
</table>